ICOT Technical Report: TR-258

TR-258

A Self Applicable Partial Evaluator and Its Use in Incremental Compilation

by
H. Fujita and K. Furukawa

May, 1987

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

A Self-Applicable Partial Evaluator and Its Use in Incremental Compilation

Hiroshi FUJITA and Koichi FURUKAWA

ICOT Research Centre
Institute for New Generation Computer Technology
1-4-28, Mita, Minato-ku, Tokyo 108 Japan

Abstract: This paper presents an experimental implementation of a self-applicable partial evaluator in Prolog used for compiler generation and compiler generator generation. The partial evaluator is an extension of a simple meta-interpreter for Prolog programs, and its self-application is straightforward because of its simplicity. A method of incremental compilation is also described as a promising application of the partial evaluator for knowledge-based systems.

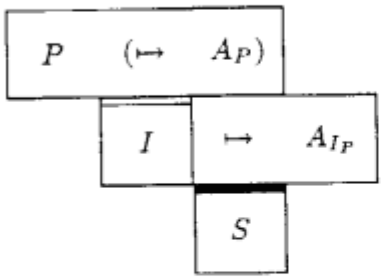
1. Introduction

The general theory and practice of partial evaluation has been well developed for software written in conventional programming languages. Recently, the same progress has been made in relatively young logic programming languages such as Prolog. In particular, partial evaluation has been shown to be a fundamental technique to make meta-programming feasible in practical use. However, there are many problems that have been proposed, but not solved yet, although some of them have already been solved in other languages or in limited contexts.

1.1. Overview of the Background

It is easy to read, write, and debug programs when meta-programming is adopted, because of the high modularity and generality obtained. However, meta-programming is frequently inefficient when the program obtained is executed, owing to the interpretation overhead (Fig.1). This flaw is remedied by using partial evaluation (Fig.2). The interpreter specialised for a program does not suffer from overhead because the program has been digested and assimilated into the body of the interpreter code. Hence, it can be executed with reasonable runtime efficiency (Fig.3).

Besides the naive usage of partial evaluation for program optimisation, a particularly interesting application to compilers and compiler generators is well known [Futamura 71], [Futamura 82] and [Ershov 78]. The specialised interpreter obtained above can

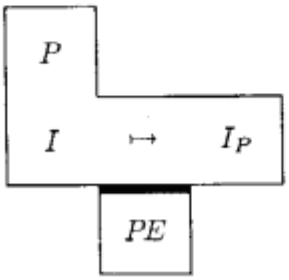


P: object-level program
I: meta-interpreter
A_P: answer to query on P
A_{I_P}: answer to query on I on P
S: system (eg. DEC-10 Prolog interpreter)

$(I : P \mapsto A_P)$

$S : I \times P \mapsto A_{I_P}$

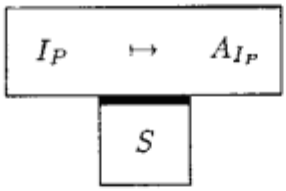
Figure 1 Interpretation overhead in meta-programming



PE: partial evaluator
I_P: specialised meta-interpreter wrt. P

$PE : I \times P \mapsto I_P$

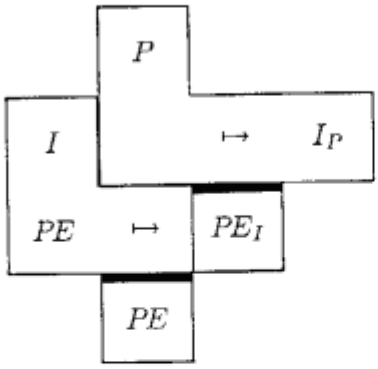
Figure 2 Partial evaluation



$S : I_P \mapsto A_{I_P}$

Figure 3 Specialised meta-interpreter running without overhead

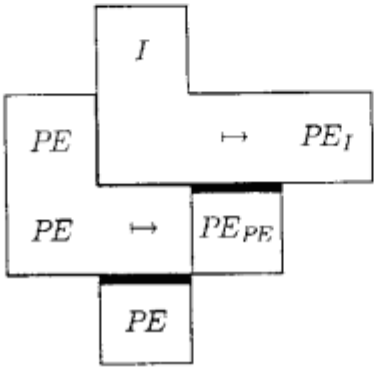
be taken as a compiled code for the object-level program, in the sense that it performs computation specific to the program faster than its source code. Hence, the partial evaluation corresponds to compilation. Stepping up one level, a partial evaluator can be partially evaluated with respect to an interpreter. The specialised partial evaluator for the interpreter can be taken as the corresponding compiler. Hence, the partial evaluation in this case corresponds to compiler generation (Fig.4). Further, a compiler generator can be generated by partially evaluating a partial evaluator with respect to another partial evaluator (Fig.5).



$PE_I : P \mapsto I_P$

$PE : PE \times I \mapsto PE_I$

Figure 4 Compiler generation and compilation



$PE_{PE} : I \mapsto PE_I$

$PE : PE \times PE \mapsto PE_{PE}$

Figure 5 Compiler generator generation and compiler generation

1.2. Relation to Other Research

The research described in this paper is an extension of [Takeuchi 86], in which the first attempt to optimise meta-programming by partial evaluation was successful. However, the next step to realise self-application of the partial evaluator, thereby performing compiler generation, compiler generator generation and incremental compilation, was not accomplished. Advancing towards this step is our main motivation.

There has been steady progress in this field of research in the Lisp community for some time. On the other hand, in the Prolog community, results are only recent, and as far as the authors know, self-application of the Prolog partial evaluator has not been successful yet. In reality, the present paper is viewed as a Prolog counterpart of, or extension of, [Jones 85] and [Sestoft 86], and largely shares its common principle. However, our partial evaluation is simpler, structurally and technically, than those of Lisp, because Prolog has better properties than Lisp in some respects, especially its binding scheme based on unification.

With regard to other research in the Prolog community, [Safra 86] and [Levi 86]

obtained results similar to [Takeuchi 86] in a slightly different context, but based on almost the same motivation, aiming at optimising meta-programming. Some general aspects of partial evaluation as program transformation and improvement are described in [Komorowski 82] and [Kursawe 86]. However, they have no special concern with meta-programming and self-application of partial evaluators.

1.3. Outline

Section 2 of this paper describes the development of a self-applicable partial evaluator in Prolog. Compilation, compiler generation and compiler generator generation using the self-applicable partial evaluator are described in section 3. The partial evaluator is extended, and thereby used in incremental compilation in section 4, following the scenario given in [Takeuchi 86]. Some performance results and evaluation are given in section 5. After another possible application is described in section 6, this paper concludes with a summary and notes on future research in section 7. The language used for programs appearing throughout the paper is DEC-10 Prolog [Pereira 79].

2. Development of the Partial Evaluator

A partial evaluator may be required to be as powerful as possible to obtain optimal codes for a wide class of source programs. Such a partial evaluator tends to become a large and complex program. On the other hand, it may be required to be as compact as possible if the user applies it to itself, that is, partially evaluates the partial evaluator using itself. The user may also require a partial evaluator somewhere between the most powerful and the most compact. The user can use a full-power partial evaluator to partially evaluate a minimal partial evaluator, thereby obtaining a specialised partial evaluator of medium size and functionality. In such a case, the partial evaluator is not necessarily required to be self-applicable. Furthermore, it may be feasible to use a language, L_1, for implementing a powerful partial evaluator which processes programs, especially a compact partial evaluator, written in another language, L_2. Such a cross-compilation-like technique is described in [Kahn 82] and [Kahn 84], where Lisp and Prolog are coupled. However, it should be worthwhile to construct a minimal but non-trivial self-applicable partial evaluator, which must be written in the same language it can recognise.

2.1. Partial Evaluation of Prolog Programs

The partial information known prior to a program execution may be given in several forms. The following two are very common in applications:

(1) Queries to the program are limited with respect to certain sets of constants or restricted patterns of data structure.

(2) The set of facts, represented as unit clauses and referred to by the program, is only partly given.

Partial information typically flows top-down in case (1) and bottom-up in case (2) respectively, but possibly alternately up and down during a computation, due to the

bi-directional nature inherent in unification.

In either case, unfolding is the most essential operation to perform constant propagation, evaluation of evaluable predicates, elimination of failing cases, and specialisation in the end.

At first, we tried to develop a powerful and automatic version of the partial evaluator in full Prolog (with extra logical features) [Fujita 87a]. However, self-application was difficult primarily for two reasons. One is that it cannot handle some features such as the *cut* operator effectively, which are used rather frequently in implementation. This is the basic flaw, but can be remedied somehow as in [Venken 84]. The other is more practical; the code of the partial evaluator itself as well as its output is so big that it is almost impossible to obtain results of a reasonable size and in a reasonable partial evaluation time. This raises the problem of space-time trade-off, in the sense of increasing code size vs. decreasing run-time of the resultant code. To solve the problem, certain criteria for evaluation of total performance must be established first. Then, the partial evaluator should be organised to work along the criteria. In any case, the partial evaluator requires some sophistication concerning program analyses, which will complicate matters when it is applied to itself.

2.2. Partial Solver

In parallel to the above research enhancing the partial evaluator and automating it, another approach was tried. A tiny partial evaluator was developed as an extension of the well known Prolog self-interpreter, `solve`, shown below.

```
solve(A)      :-  solve_prim(A).
solve((A,B)) :-       solve(A),     solve(B).
solve(A)      :-      clause(A,B),   solve(B).
```

The standard Prolog interpreter solves the goal, `solve(A)`, with the same result as if it solves the goal, `A`, directly. `solve_prim(A)` evaluates goal `A` if it is `true` or any other primitive defined in the system. `clause(A,B)` succeeds if `A` is unified with a head of clause accessible in the system, unifying `B` with its body, otherwise it fails.

This simple definition of a Prolog self-interpreter, `solve`, suggests the following *partial solver*, `psolve`.

```
psolve(A,R)             :- psolve_prim(A,R).
psolve((A,B),(RA,RB)) :-      psolve(A,RA), psolve(B,RB).
psolve(A,R)             :-      clause(A,B),  psolve(B,R).
psolve(A,A)             :-   suspended(A).
```

The partial solver, `psolve`, partially solves a given goal, `A`, with the result of `R` which is a conjunction of subgoals under `A` suspended to be solved. `R` is called *residual goal(s)* for `A`. A goal yields residual goals if it is a primitive and partially evaluated by `psolve_prim` with the result of the residual goals, or if it is a conjunction and some of the conjunct yields residual goals, or if it is of a user-defined predicate whose definition

can be accessed by `clause` and the subgoals yield residual goals, or if the goal itself should be suspended for some reason.

Note that `psolve` is related to `solve` in the following clause:

```
solve(A) :- psolve(A,R), all_true(R).
```

where `all_true(R)` succeeds when `R` is a conjunction all of whose conjuncts are `true`. Thus, if `A` is `psolved` with no residual goal other than `true`,it is, in fact, solved totally. Conversely, it can be said that if `A` is solvable at `psolve`-time, it is, in fact, `psolved` with no residual goals; however, even if it is not solvable, `psolve` will yield residual goals for it. In a sense, `psolve` covers only successful sub-proof-trees from the root goal, hence, the name "partial solver".

2.3. Goal Suspension

What goal should be suspended and made residual? In general, the expansion of a goal whose arguments are not fully instantiated may be of no use since it may not reduce the real computation. Even worse, the expansion of such a goal would cause infinite expansions of subgoals, making the process nonterminate.

For instance, consider the definition of `append` given as:

```
append([H|X],Y,[H|Z]) :- append(X,Y,Z).
append([],Y,Y).
```

The goal, `append([1,2],Y,Z)`, can be expanded finitely, or solved totally with the resultant substitution `Z=[1,2|Y]`. Also the goal, `append(X,Y,[1,2])`, is solved with `{X=[], Y=[1,2]}` or `{X=[1], Y=[2]}` or `{X=[1,2], Y=[]}`. However, the goal, `append(X,Y,Z)`, cannot be solved finitely, hence, it should be suspended by `psolve`. The suspension will be forced if the following clause is supplied.

```
suspended(append(A,B,C)) :- var(A), var(C).
```

That is, if the goal in hand is `append(A,B,C)` and both arguments `A` and `C` are unbound, it is to be suspended. Thus, for example, `append([1,2|X],Y,Z)` should be `psolved` up to `append(X,Y,W)` with the substitution, `Z=[1,2|W]`, and `append(X,[],[1,2|Z])` should be `psolved` up to `append(U,[],Z)` with the substitution `X=[1,2|U]`.

In general, a goal of recursive predicate must be unfolded very carefully. The safest way, although it might be too conservative, is to find some well-founded-ordering (WFO) in the sequence of consecutive recursive calls.

$$p(\mathbf{x}) \succ p(\mathbf{x}') \succ p(\mathbf{x}'') \succ \ldots$$

For one such WFO which is often applicable to usual applications, the subterm relation can be used. Suppose a goal in hand, $p(\mathbf{x})$, matches the heads of its defining clauses $\{p(\mathbf{t}) :- \ldots p(\mathbf{s}) \ldots\}$, by the substitution σ for the variables in $\mathbf{t}$ such that $\mathbf{t}\sigma \equiv \mathbf{x}$ for each $\mathbf{t}$. Then, all the recursive calls in the body of the clauses above are $\{p(\mathbf{x}')\}$ such

that $s\sigma \equiv x'$, for each s. Now, if each of the x' is a proper subterm of x, the unfolding is safely performed. If x is a vector of arguments, the above condition should be checked at least at one fixed position in the vector.

Although there may be other WFO more complex than the subterm relation, it seems difficult to find every possibility. Only a few of them might be mechanically found by analysing the program clauses relevant to the given goal. The elaboration of this issue is beyond the scope of the present paper.

2.4. Partial Solver as A Partial Evaluator

After the residual goal `R` is obtained by `psolve`, the solving process can be resumed for `A` by solving `R`, provided that the condition is changed in preference for `A` to be solved further. In a sense, `R` is a (possibly conjunction of) subgoal(s) on the "snapshot of a wavefront" in the proof tree for `A`, the root of which is labelled with `A`. At resumption, the process can be restarted at the wavefront instead of at the root, as if `R` were the immediate subgoal(s) for `A`. Accordingly, `A:-R` can be considered as a program clause, and is called a *residual program clause*. Further, the residual program clause, `A:-R`, is taken as a partially evaluated program clause for `A`, and `psolve` is taken as a kind of partial evaluator. A similar idea is described in [Vasey 86], where *qualified answer* corresponds to our residual program clause.

Now, turning to the suspension mechanism, it should be noticed that the above `psolve` is in danger of choosing the third instead of the fourth clause, regardless of the `residual` condition. The decision on which clause should be chosen must be made outside `psolve`. Therefore, the structure of `psolve` is reformed as follows:

```
psolve(A,R)        :- psolve_prim(A,R).                          ...(P1)
psolve((A,B),U-W)  :-     psolve1(A,U-V), psolve1(B,V-W).       ...(P2)
psolve(A,R)        :-          cl(A,B),   psolve1(B,R).         ...(P3)

psolve1(A,R)       :-  expandable(A),       psolve(A,R).        ...(P4)
psolve1(A,[A|Z]-Z) :-    residual(A).                           ...(P5)
```

Clause (P1) is the same as before. Now, at (P2), if the goal to `psolve` is a conjunction, then the residual goal for it is given by appending the residual goals for each conjunct given by `psolve1` instead of `psolve`. The conjunction of the residual goals is represented by *d-list**. At (P3), if the goal is a literal and user defined, then its definition is retrieved by `cl`, in place of `clause`, and its residual goal is given by `psolve1`, instead of `psolve`, applied to the body goal of the selected clause. `cl` is almost the same as `clause` except that it is intended to retrieve only the subject clauses selectively, when they are processed by `psolve`.

The newly introduced predicate `psolve1` checks the expandability of goal `A` and

* For example, the list of two elements, [1,2] is represented as `[1,2|Z]-Z`, which is easily appended to another d-list, say, `[3,4|W]-W`, obtaining `[1,2,3,4|W]-W`, simply by unifying the unbound tail of the first, `Z`, with the head of the other d-list.

passes it to psolve if it is expandable at (P4), otherwise psolve1 suspends it at (P5). If A is residual, then it may not be expandable. Thus, a goal is added to the d-list if and only if it is residual. Any goal that should be suspended cannot be expanded by its defining clauses.

The new predicates, expandable and residual, are defined as:

```
expandable(A) :- \+$suspend(A).
  residual(A) :-   $suspend(A).
```

where \+ is the DEC-10 Prolog *not* operator which is interpreted by the *negation as failure* rule. $suspend is essentially the same as suspended in the previous definition of psolve, whereas it is marked by $ at the head of the predicate name with the intention of indicating that the condition will be given by the user. For example, the user can provide the clause:

```
$suspend(append(A,_,C)) :- var(A), var(C).
```

It seems redundant to have both expandable and residual, which are complements of each other; however, both are introduced so as to define psolve1 without using negation (or *cut*, *if-then-else*, etc.).

Observe that all of the clauses for psolve and psolve1 are of disjoint cases. The only nondeterminacy is caused by cl in (P3), which contributes alternative results for the residual goal and the residual program clauses. By backtracking the goal, psolve(A), all alternatives of the residual program clauses for A can be collected. Therefore, the top-level command for the partial evaluator using psolve can be introduced as:

```
psolve_all(A) :- bagof((A:-R),psolve(A,R),NewCls),
                                        define(NewCls).
```

The command psolve_all collects all the alternatives for the residual goal, R, using the DEC-10 Prolog bagof primitive*. Then, it defines the residual program clauses, {A:-R}, for A. define(NewCls) asserts each of the resultant clauses in the list NewCls so that it is accessible by cl when it is further processed.

It should be noted that psolve_all is not considered as a part of the program for the partial evaluator; it is just a command. Accordingly, bagof and define are considered command primitives rather than the primitive predicates for psolve program, and never appear in the residual program clauses when psolve is applied to itself.

In principle, once the residual program clauses for A are obtained, any instance of A can be solved by using the residual program clauses instead of the original program clauses for A. It is important to mention, however, that some of the goals appearing in the residual program clauses may still need to be solved by the original program

* bagof(A,P,S) computes all solutions for A satisfying condition P with the resultant list of solutions S, if at least one solution exists. Otherwise, it fails.

clauses. In particular, for recursively defined predicates, both residual and original program clauses may be needed.

2.5. Partial Evaluation of Primitives

The primitive predicates are those defined as the DEC-10 Prolog system predicates plus `psolve_prim`, `cl`, `expandable` and `residual`. The predicates in the latter group are called *pseudo-primitives* to distinguish them from the other genuine primitives in the system, although they may be regarded as just primitives in the sequel.

The primitives are defined once and for all for their partial evaluation as well as normal evaluation. For instance, the partial evaluation of `true` and DEC-10 Prolog `is` and `<` primitives are defined as follows:

Partial Evaluation of `true`, `is` and `<`

```
psolve_prim(true,Z-Z)          :- !.

psolve_prim(A is B,Z-Z)        :-   ground(B),  !, call(A is B).
psolve_prim(A is B,R)          :- simplify(B,C),!,   is(A,C,R).

          is(A,C,Z-Z)          :-      var(C),  !, A=C.
          is(A,C,[A is C|Z]-Z).

psolve_prim(A<B,Z-Z)           :- ground(A<B),  !, call(A<B).
psolve_prim(A<B,[A<B|Z]-Z)     :- !.
```

Where `ground(X)` succeeds when `X` is a ground term, ie. it is an atomic constant or a compound term constructed only with function symbols and constants, or in short, has no variables. `simplify(X,Y)` succeeds when `X` is a legal numerical expression, returning a simplified expression, `Y`, which will possibly be identical to `X`. The detail of the inner definition of `simplify` is omitted. The standard interpreter of DEC10-Prolog solves the `call(X)` primitive by taking the term, `X`, as a predication, thereby solving `X` as if it were an immediate goal to be solved.

The partial evaluation of pseudo-primitives are defined in the following.

Partial Evaluation of `cl`

```
psolve_prim(cl(A,B),[cl(A,B)|Z]-Z) :- var(A),!.
psolve_prim(cl(A,B),          Z-Z) :-  cl(A,B).
```

Partial evaluation of `cl` should make it residual if its first argument, `A`, is unbound, otherwise, it is solved as in normal evaluation.

Partial Evaluation of `expandable` and `residual`

```
psolve_prim(expandable(A),[expandable(A)|Z]-Z) :-        var(A),!.
psolve_prim(expandable(A),                 Z-Z) :- expandable(A),!.
```

```
psolve_prim(residual(A),[residual(A)|Z]-Z)      :-        var(A),!.
psolve_prim(residual(A),                Z-Z)    :-   residual(A),!.
```

Partial evaluation of these follows the same principle as that for `cl`. That is, if `expandable(A)` (`residual(A)`) is given a variable, ie. an unknown inner goal, `A`, then it must be made residual at partial evaluation time, otherwise, it is solved immediately in the present context as in the normal evaluation mode.

An explicit unification, `X=Y`, is always evaluated and never made residual. Further, the partial evaluation of a (pseudo-)primitive never makes residual goals like `var(_)`, `\+_`, `!`(cut), `ground(_)`, and `call(_)` and `_==_` used in the definitions. They are taken as more elemental than the primitives, and hidden within the black box of the `psolve_prim` semantics. Accordingly, they are never specialised or exposed in the residual goals even when the whole partial evaluator is applied to itself.

Now, if goal `A` is `psolved` to be `true` then `psolve(A,R)` is also `psolved` to be `true`, with the result, `R=Z-Z` (null d-list), that is:

```
psolve(psolve(A,Z-Z),Y-Y) :- psolve(A,V-W), V==W.
```

In general, when partial evaluation of a goal, `A`, gives a conjunction of residual goals $R_1, R_2, \ldots, R_n$, partial evaluation of `psolve(A,R)` should give the conjunction of `psolve(`R_i `,...)`. That is, the call `psolve(psolve(A,R),R')` should result in:

```
psolve(psolve(A, U1-W),
       [psolve(R1,U1-U2),psolve(R2,U2-U3),...,psolve(Rn,Un-W)|Z]-Z)
```

This is coded in `psolve_prim` more precisely below.

Partial Evaluation of `psolve_prim`

```
psolve_prim(psolve_prim(A,R),[psolve_prim(A,R)|Z]-Z) :- var(A),!.
psolve_prim(psolve_prim(A,Q),R) :-
            psolve_prim(A,P), pprim(P,Q,R).

        pprim(A-C,W-W,Z-Z) :- A==C,!.
        pprim([A|B]-C,U-W,[psolve_prim(A,U-V)|Y]-Z) :-
            pprim(B-C,V-W,Y-Z),!.
```

The same trick could be used for `psolve`, however, it would turn out nothing but hand-writing of the compiler generator. This must be avoided in principle, although it seems inevitable for primitives such as `psolve_prim`, because it is against the motivation of self-application, ie. mechanical (hopefully automatic) generation of compilers and compiler generators only by the partial evaluator of the most general and minimal structure.

The problem of what goal of `psolve(A,...)` should be suspended and made residual at the time of partial evaluation depends on inner goal `A`, and is controlled

only by appropriate `$suspend` conditions which are accessible through `expandable` and `residual`.

3. Compiler and Compiler Generator

The partial evaluator `psolve` is used in compilation, compiler generation and compiler generator generation.

3.1. Compilation

Suppose that we have the following `int` as a meta-interpreter.

```
int(true,[100]).                                                    ...(I1)
int((A,B),Z)       :-  int(A,X), int(B,Y), append(X,Y,Z).           ...(I2)
int(not(A),[CF])   :-  int(A,[C]), C < 20, CF is 100-C.             ...(I3)
int(A,[CF])        :- rule(A,B,CF1), int(B,S), cf(CF1,S,CF).        ...(I4)

cf(X,Y,Z)          :- product(Y,100,W), Z is (X*W)/100.             ...(I5)

product([A|X],Y,Z)     :- W is A*Y/100, product(X,W,Z).             ...(I6)
product([],Y,Y).                                                    ...(I7)

append([A|X],Y,[A|Z]) :- append(X,Y,Z).                             ...(I8)
append([],Y,Y).                                                     ...(I9)
```

This is a tiny inference engine for rules with a certainty factor. Now, suppose also that we are given the following `rule` as an object-level program to `int`.

```
rule(should_take(Person,Drug),                                      ...(R1)
   ( complains_of(Person,Symptom),
     suppresses(Drug,Symptom),
     not(unsuitable(Drug,Person)) ),                    70).

rule(unsuitable(Drug,Person),                                       ...(R2)
   ( aggravates(Drug,Condition),
     suffers_from(Person,Condition) ),                  80).

rule(suppresses(aspirin,pain),                          true, 60).  ...(R3)
rule(aggravates(aspirin,peptic_ulcer),                  true, 70).  ...(R4)

rule(suppresses(lomotil,diarrhoea),                     true, 65).  ...(R5)
rule(aggravates(lomotil,impaired_liver_function), true, 70).  ...(R6)
```

Then, `int` is specialised with respect to `rule`, provided with the control information:

```
$suspend(int(A,_))        :- var(A) ;
          inst(A,complains_of(_,_)) ; inst(A,suffers_from(_,_)).
```

```
$suspend(product(A,_,_)) :- var(A).
$suspend( append(A,_,C)) :- var(A), var(C).
```

Where `inst(A,B)` is a primitive, and succeeds if `A` is an instance of `B`, ie. $B\sigma$ becomes identical to `A` with some substitution σ for variables in `B`, and fails otherwise. The goals, `int(complains_of(_,_),_)` and `int(suffers_from(_,_),_)`, are suspended because they will be given only when the program is executed.

The compilation is performed by the command:

```
:- psolve_all(int(should_take(_,_),_)).
```

The specialised interpreter, $\texttt{int}_{\texttt{rule}}$, is obtained as:

```
int(should_take(A,aspirin),[B]) :-                              ...(Ir1)
   int(complains_of(A,pain),C),
   int(suffers_from(A,peptic_ulcer),D),
   product(D,70,E),
   F is 80*E/100,
   F<20,
   G is 100-F,
   append(C,[60,G],H),
   product(H,100,I),
   B is 70*I/100.

int(should_take(A,lomotil),[B]) :-                              ...(Ir2)
   int(complains_of(A,diarrhoea),C),
   int(suffers_from(A,impaired_liver_function),D),
   product(D,70,E),
   F is 80*E/100,
   F<20,
   G is 100-F,
   append(C,[65,G],H),
   product(H,100,I),
   B is 70*I/100.
```

The code for $\texttt{int}_{\texttt{rule}}$ has been fully specialised for the queries to the object-level program, `should_take(Person,aspirin)` and `should_take(Person,lomotil)`.

3.2. Compiler Generation

The partial evaluator, `psolve`, is used to specialise itself with respect to the meta-interpreter `int`, obtaining $\texttt{psolve}_{\texttt{int}}$, which in turn is used to obtain the specialised meta-interpreter $\texttt{int}_{\texttt{rule}}$ as shown in the last subsection.

The control information is given as:

```
$suspend(psolve1(A,_))    :- var(A) ; residual(A).
```

```
$suspend(    int(A,_))    :- var(A).
$suspend(product(A,_,_)) :- var(A).
$suspend( append(A,_,C)) :- var(A), var(C).
$suspend(   rule(_,_,_)).
```

Compiler generation is performed by the command:

```
:- psolve_all(psolve(int(_,_),_)).
```

The compiler, $\texttt{psolve}_{\texttt{int}}$, is obtained as:

```
psolve(int(true,[100]),A-A).                                    ...(Pi1)
psolve(int((A,B),C),D-E) :-                                     ...(Pi2)
       psolve1(int(A,F),D-G),
       psolve1(int(B,H),G-I),
       psolve1(append(F,H,C),I-E).
psolve(int(not(A),[B]),C-D) :-                                  ...(Pi3)
       psolve1(int(A,[E]),C-F),
   psolve_prim(E<20,F-G),
   psolve_prim(B is 100-E,G-D).
psolve(int(A,[B]),C-D) :-                                       ...(Pi4)
       psolve1(rule(A,E,F),C-G),
       psolve1(int(E,H),G-I),
       psolve1(product(H,100,J),I-K),
   psolve_prim(B is F*J/100,K-D).
```

Each of the $\texttt{psolve}_{\texttt{int}}$ clauses reflects the corresponding clause of the `int` program. The clause (Pi1) means that `int(true,[100])` is always partially evaluated with the null residual goal, or is totally evaluated as `true`. The clause (Pi2) means that the partial evaluation of `int((A,B),C)` gives the conjunction of residual goals, each of which is given by partially evaluating `int(A,F)`, `int(B,H)` and `append(F,H,C)` respectively. The other clauses, (Pi3) and (Pi4), should be read similarly. The non-recursive `cf` clause has been absorbed into (Pi4).

Note that in addition to the main predicate `int`, the other two recursive predicates, `product` and `append`, should be taken in by the compiler, because they will be called from within the compiler. Thus, the following is added.

```
:- psolve_all(psolve(product(_,_,_),_)),
   psolve_all(psolve( append(_,_,_),_)).

psolve(product([A|B],C,D),E-F) :-                               ...(Pi5)
   psolve_prim(G is A*C/100,E-H),
   psolve1(product(B,G,D),H-F).
psolve(product([],A,A),B-B).                                    ...(Pi6)
psolve(append([A|B],C,[A|D]),E) :-                              ...(Pi7)
   psolve1(append(B,C,D),E).
```

```
psolve(append([],A,A),B-B).                                    ...(Pi8)
```

What will happen if one of the item of control information is not enough? For instance, suppose that one of them is changed to:

```
$suspend(psolve1(A,_)) :- var(A).
```

instead of:

```
$suspend(psolve1(A,_)) :- var(A) ; residual(A).
```

Then, the resultant code will become:

```
psolve(int(true,[100]),A-A).
psolve(int((A,B),C),[int(A,D),int(B,E),append(D,E,C)|F]-F).
psolve(int(not(A),[B]),[int(A,[C])|D]-E) :-
   psolve_prim(C<20,D-F),
   psolve_prim(B is 100-C,F-E).
psolve(int(A,[B]),[rule(A,C,D),int(C,E),product(E,100,F)|G]-H) :-
   psolve_prim(B is D*F/100,G-H).
psolve(product([A|B],C,D),E-F) :-
   psolve_prim(G is A*C/100,E-[product(B,G,D)|F]).
psolve(product([],A,A),B-B).
psolve(append([A|B],C,[A|D]),[append(B,C,D)|E]-E).
psolve(append([],A,A),B-B).
```

The second clause means that partial evaluation of a goal `int((A,B),C)` will always give the conjunction of the goals, `int(A,D)`, `int(B,E)` and `append(D,E,C)`, immediately as its residual goals. No evaluation or unfolding will be performed, even if the goals are instantiated enough to be evaluated or unfolded when the compiler is executed. This result is not intentional. It is due to the inappropriate time of decision made on the residual goals, ie. earlier than the correct time. In fact, the most delicate task is to have exact control information to continue or stop partial evaluation appropriately.

3.3. Compiler Generator Generation

The partial evaluator, `psolve`, is used to specialise itself with respect to itself, obtaining $\texttt{psolve}_{\texttt{psolve}}$, which in turn is used to obtain the compiler, $\texttt{psolve}_{\texttt{int}}$, as shown in the last subsection.

The compiler generator generation is performed by the command:

```
:- psolve_all(psolve(psolve(_,_),_)).
```

The compiler generator, $\texttt{psolve}_{\texttt{psolve}}$, is obtained as:

```
psolve( psolve(A,B),C) :-                                      ...(Pp1)
   psolve_prim(psolve_prim(A,B),C).
```

```
psolve( psolve((A,B),C-D),E-F) :-                          ...(Pp2)
       psolve1(psolve1(A,C-G),E-H),
       psolve1(psolve1(B,G-D),H-F).
psolve( psolve(A,B),C-D) :-                                ...(Pp3)
   psolve_prim(cl(A,E),C-F),
       psolve1(psolve1(E,B),F-D).
psolve(psolve1(A,B),C-D) :-                                ...(Pp4)
   psolve_prim(expandable(A),C-E),
       psolve1(psolve(A,B),E-D).
psolve(psolve1(A,[A|B]-B),C-D) :-                          ...(Pp5)
   psolve_prim(residual(A),C-D).
```

This is quite simple, and even seems to be trivial; however, it is enough to handle `psolve(psolve...)` anyway. Note that the original clauses, (P1)...(P5), are needed for the compiler generator to work as a whole.

Incidentally, it would be interesting to compare this to the self-interpreter of pure Prolog, `solve`, defined as follows:

```
solve(true).
solve((A,B)) :-        solve(A),          solve(B).
solve(A)     :-       clause(A,B),        solve(B).
```

What will happen if the goal, `solve(solve(A))`, is given to the standard pure Prolog interpreter? The result would be the same as when the goal, `solve(A)`, or even `A` is given directly. In this sense, it can be said that `solve` is idempotent. This interesting property can be shown by partial evaluation as follows:

```
$suspend( solve(A))   :- var(A).
$suspend(clause(A,_)) :- var(A).

:- psolve_all(solve(solve(A))).

  solve(solve(true)).
  solve(solve((A,B))) :- solve( solve(A)),   solve(solve(B)).
  solve(solve(A))     :- solve(clause(A,B)), solve(solve(B)).
```

At this point, if we assume:

```
solve(clause(A,B)) :- clause(A,B).
```

and, applying the replacement `{solve2(A) / solve(solve(A))}` at each atom, then the following result is obtained:

```
solve2(true).
solve2((A,B)) :-       solve2(A),         solve2(B).
solve2(A)     :-       clause(A,B),       solve2(B).
```

Thus, the `solve2` program, which is derived as the residual program for the special goal, `solve(solve(A))`, is isomorphic to the original `solve` program. Now, let us compare this with the result of the compiler generator obtained above. It would be possible to replace the atoms in the clauses as:

```
     psolve2((A,B),C) /           psolve(psolve(A,B),C)
psolve_prim2((A,B),C) / psolve_prim(psolve_prim(A,B),C)
   psolve1_2((A,B),C) /         psolve1(psolve1(A,B),C)
         cl2((A,B),C) /          psolve_prim(cl(A,B),C)
```

obtaining the reformed program:

```
psolve2((A,B),C)          :- psolve_prim2((A,B),C).
psolve2(((A,B),C-D),E-F) :-    psolve1_2((A,C-G),E-H),
                               psolve1_2((B,G-D),H-F).
psolve2((A,B),C-D)        :- cl2((A,E),C-F), psolve1_2((E,B),F-D).
```

Obviously, it is not isomorphic to the original `psolve` program. `solve2` was another interpreter in the same level as `solve`. On the other hand, `psolve2` is not another partial evaluator in the same level as `psolve`; it is essentially a different partial evaluator, being one level higher than `psolve`. However, it can collapse to be the partial evaluator isomorphic to the lower `psolve` in its behaviour, when it gives a null residue, ie.:

```
psolve(A,R) :- psolve2((A,R),X-Z), X==Z.
```

4. Incremental Compilation

So far, the object-level program is assumed to be given as a whole at one time. Now, suppose that, say, (R3)...(R6) in section 3.1 or facts about other medicines are given only one by one ocasionally. Then, can we do anything with the already known `int` and partial `rule` at hand? According to the principle of partial evaluation, something must be done even in such circumstances. To do this, however, the `psolve` used so far should be extended somehow to manage this new situation.

4.1. Open Predicate

The predicate where not all of the defining clauses are given is said to be *open*. For example, consider the following `ancestor` program:

```
ancestor(A,B) :- parent(A,B).
ancestor(A,B) :- parent(A,C), ancestor(C,B).
```

where two `parent` relations are given as:

```
parent(p,q).
parent(q,r).
```

If the parent relation is closed, ie. it is not open, then the ancestor program is computed immediately and simply reduced to:

```
ancestor(p,q).
ancestor(q,r).
ancestor(p,r).
```

This is a solution table rather than a program, in the sense that it needs no more computation (inference) than looking up the list of facts. However, if the parent is still open, then the ancestor program should be partially evaluated to:

```
ancestor(p,q).
ancestor(q,r).
ancestor(A,B) :- parent(A,B).
ancestor(p,B) :- ancestor(q,B).
ancestor(q,B) :- ancestor(r,B).
ancestor(A,B) :- parent(A,C), ancestor(C,B).
```

The two parent relations have been absorbed into the ancestor clauses. The remaining calls to parent in the third and the last clauses of the specialised ancestor program are required to refer to other facts about parent which will be given in future. In general, the specialised program with open predicates includes the original clauses as well as those that have assimilated the known facts.

Thus, if the goal is of an open predicate, it is suspended somewhere in the specialised clauses. This is the essential mechanism to realise incremental compilation described in the sequel.

4.2. Extended Partial Solver

The partial solver psolve is extended to handle open predicates.

```
 psolve(A,R)          :-          cl(A,B),   psolve1(B,R).      ...(PE1)

psolve1((A,B),X-Z)  :-      psolve2(A,X-Y), psolve1(B,Y-Z).   ...(PE2)
psolve1(A,R)         :-      literal(A),     psolve2(A,R).     ...(PE3)

psolve2(A,R)         :- psolve_prim(A,R).                      ...(PE4)
psolve2(A,R)         :-         open(A),      psolve(A,R).     ...(PE5)
psolve2(A,R)         :-      psolve3(A,R).                     ...(PE6)

psolve3(A,R)         :-  expandable(A),       psolve(A,R).     ...(PE7)
psolve3(A,[A|Z]-Z)   :-    residual(A).                        ...(PE8)
```

This also reflects changes in structure for optimisation purposes under the assumptions that the top-level query to the psolve is always a literal and the body literals returned by cl in B are a right-linear conjunction, $(B_1,(B_2,\ldots(B_{n-1},B_n)\ldots))$.

Two new primitives are introduced: `literal` and `open`. `literal(A)` succeeds if `A` is a literal, ie. not a conjunction, otherwise it fails. `open(A)` succeeds if `A` is a goal of an open predicate discussed above, otherwise it fails. The user should inform whether a predicate is open or not by `$open`, thus:

```
open(A) :- $open(A).
```

Moreover, the primitives `expandable` and `residual` are redefined as:

```
expandable(A) :- $expand(A).
  residual(A) :-  \+prim(A), (\+$expand(A) ; $open(A)).
```

Where another inner primitive, `prim(A)`, succeeds if `A` is of a primitive predicate (including pseudo-primitive), otherwise it fails.

Now, the `$suspend` in old definition is replaced by `$expand`. For example,

```
$suspend(append(A,_,C)) :-    var(A),    var(C).
```

is now replaced by:

```
 $expand(append(A,_,C)) :- nonvar(A); nonvar(C).
```

That is, conditions for `suspend` and `expand` are complements of each other.

Note that if goal `A` is of an open predicate having some defining clauses at partial evaluation time, its partial evaluation calls `cl(A...)` through (PE5) and (PE1) obtaining the definition for each clauses already defined, whereas `A` will be made still residual at (PE8).

4.3. Compiling Program Fragments

Starting with the compiler generated by the extended `psolve` and `int` in a similar way to that described in section 3.2, incremental compilation can be performed by specialising the interpreter with respect to program fragments given incrementally (Fig.6).

For instance, suppose (R1) and (R2) from the `rule` definition in section 3.1 are given as the first fragment. Then, `int` $_{(R1+R2)}$ is obtained by:

```
$expand( psolve(A,_))   :- nonvar(A).
$expand(psolve1(A,_))   :- nonvar(A).
$expand(psolve2(A,_))   :- nonvar(A),
                           (prim(A) ; open(A) ; expandable(A)).
$expand(psolve3(A,_))   :- nonvar(A), (prim(A) ; expandable(A)).

$expand(    int(A,_))   :- nonvar(A),
         \+inst(A,complains_of(_,_)), \+inst(A,suffers_from(_,_)).
$expand(     cf(_,_,_)).
```

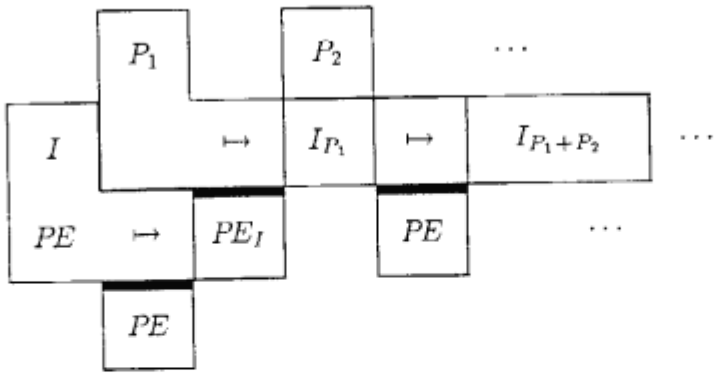


Figure 6 Incremental compilation

```
$expand(product(A,_,_)) :- nonvar(A).
$expand( append(A,_,C)) :- nonvar(A) ; nonvar(C).

$open(     rule(_,_,_)).

:- psolve_all(psolve(      int(_,_),_)).
```

The result is:

```
int(should_take(A,B),[C]) :-
   int(complains_of(A,D),E),
   rule(suppresses(B,D),F,G),
   int(F,H),
   product(H,100,I),
   J is G*I/100,
   rule(aggravates(B,K),L,M),
   int(L,N),
   product(N,100,O),
   P is M*O/100,
   int(suffers_from(A,K),Q),
   product(Q,P,R),
   S is 80*R/100,
   S<20,
   T is 100-S,
   append(E,[J,T],U),
   product(U,100,V),
   C is 70*V/100.
```

Suppose, (R3) and (R4) are given next then, `int (R1+R2)` is further specialised to `int (R1+R2+R3+R4)` by:

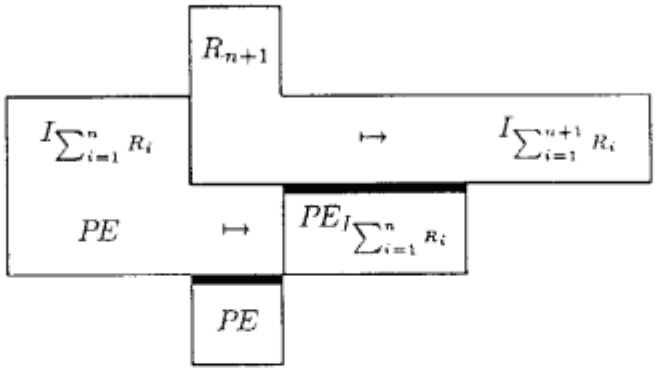


Figure 7 Compiler at stage n and its use

```
$expand(rule(_,_,_)).

:- psolve_all(int(should_take(_,_),_)).
```

The result is (Ir1) in section 3.1.

In the same way, if (R5) and (R6) are given, (Ir2) is obtained. Further, each time facts about any other medicine are given, compilation can be performed similar to that above.

At this point, it should be noted that it is better to make `psolve int (R1+R2)` first, then use it to compile facts about medicines, instead of using the bare `psolve`, and `int (R1+R2)`. In general, when an additional program fragment, R_{new}, is given, a partial evaluator (PE) has to run both on $I_{\sum R}$ and R_{new}. On the other hand, the specialised partial evaluator, $PE_{I_{\sum R}}$, or the specialised compiler, $C_{\sum R}$, need run only on R_{new} (Fig.7). Clearly, the specialised partial evaluator, which has already digested all of the raw materials given till that time, runs faster than the bare partial evaluator, which has to operate from scratch.

To obtain `psolve int (R1+R2)`, `$open(rule(_,_,_))` in the setting obtaining `int (R1+R2)`, is replaced by:

```
$open(rule(A,_,_)) :- inst(A,suppresses(_,_)) ;
                      inst(A,aggravates(_,_)).
```

Now, `rule` is not open in its most general goal pattern `rule(_,_,_)`. However, two instantiated goal patterns, `rule(suppresses...)` and `rule(aggravates...)`, are still open.

The result is as follows:

```
psolve(int(should_take(A,B),[C]),D-E) :-
       psolve2(int(complains_of(A,F),G),D-H),
```

```
    psolve3(rule(suppresses(B,F),I,J),H-K),
    psolve2(int(I,L),K-M),
    psolve2(product(L,100,N),M-O),
psolve_prim(P is J*N/100,O-Q),
    psolve3(rule(aggravates(B,R),S,T),Q-U),
    psolve2(int(S,V),U-W),
    psolve2(product(V,100,X),W-Y),
psolve_prim(Z is T*X/100,Y-A1),
    psolve2(int(suffers_from(A,R),B1),A1-C1),
    psolve2(product(B1,Z,D1),C1-E1),
psolve_prim(F1 is 80*D1/100,E1-G1),
psolve_prim(F1<20,G1-H1),
psolve_prim(I1 is 100-F1,H1-J1),
    psolve2(append(G,[P,I1],K1),J1-L1),
    psolve2(product(K1,100,M1),L1-N1),
psolve_prim(C is 70*M1/100,N1-E).
```

Note that it is enough to have only the specialised compiler, instead of the specialised interpreter, provided that the compiler at each stage is used many times. This process is extended to any stage up to n in the same way. At any stage, n, the specialised compiler can be used either to compile new program fragment, or to obtain compiled code for the program fragments accumulated within the compiler till that time, by running it with null input.

5. Performance Analyses

This section describes some performance results and their evaluation. All data is collected for the extended partial solver, `psolve`, defined in section 4.2.

Table 1 shows the results in compilation, compiler generation and compiler generator generation, where code size is figured by the number of body literals summed up for all the clauses, vs. the number of clauses, which are relevant to the corresponding program mentioned in the first column of the table. The implicit `true` primitive of a unit clause is counted as one.

Compilation of the object-level program, `rule` (R), by using the compiler $\texttt{psolve}_{\texttt{int}}$ (PE_I), took about 3/4 of the time spent by bare `psolve` (PE). Although extra time is required to generate PE_I, it should become negligible if it is used many times for different object-level program. The run-time generating $\texttt{psolve}_{\texttt{psolve}}$ (PE_{PE}) is small because PE is so simple.

Table 2 shows the results in incremental compilation. An incremental specialisation of the meta-interpreter, `int` (I), may proceed as:

$$I \Rightarrow I_{(R_{1,2})} \Rightarrow I_{(R_{1,2,3,4})}$$

which amounts to 1.18 (= 0.84 + 0.34) seconds of the total run-time, whereas another

Table 1 Compilation, compiler generation and compiler generator generation

Operation	Time (sec)	Code Size $\left(\frac{\sum(\#of\ body\ literals)}{\#of\ clauses}\right)$
$PE : I \times R \mapsto I_R$	1.27	$\frac{13}{8} : \frac{17}{9} \times \frac{6}{6} \mapsto \frac{18}{2}$
$PE : PE \times I \mapsto PE_I$	1.07	$\frac{13}{8} : \frac{13}{8} \times \frac{17}{9} \mapsto (\frac{13+}{8+})\frac{16}{8}$
$PE_I : R \mapsto I_R$	0.88	$\frac{13+16}{8+8} : \frac{6}{6} \mapsto \frac{18}{2}$
$PE : PE \times PE \mapsto PE_{PE}$	0.89	$\frac{13}{8} : \frac{13}{8} \times \frac{13}{8} \mapsto (\frac{13+}{8+})\frac{17}{12}$

PE: extended `psolve` (PE1 ... PE8)
I: `int` (I1 ... I9)
R: `rule` (R1 ... R6)

Table 2 Incremental compilation

Operation	Time (sec)	Code Size $\left(\frac{\sum(\#of\ body\ literals)}{\#of\ clauses}\right)$
$PE : I \times (R_{1,2}) \mapsto I_{(R_{1,2})}$	0.84	$\frac{13}{8} : \frac{17}{9} \times \frac{2}{2} \mapsto (\frac{17+}{9+})\frac{17}{1}$
$PE_I : (R_{1,2}) \mapsto I_{(R_{1,2})}$	0.50	$\frac{13+16}{8+8} : \frac{2}{2} \mapsto (\frac{17+}{9+})\frac{17}{1}$
$PE_{I_{(R_{1,2})}} : \phi \mapsto I_{(R_{1,2})}$	0.19	$\frac{13+16+17}{8+8+1} : 0 \mapsto (\frac{17+}{9+})\frac{17}{1}$
$PE : PE \times I_{(R_{1,2})} \mapsto PE_{I_{(R_{1,2})}}$	1.93	$\frac{13}{8} : \frac{13}{8} \times \frac{17+17}{9+1} \mapsto (\frac{13+}{8+})\frac{16+17}{8+1}$
$PE : PE_I \times (R_{1,2}) \mapsto PE_{I_{(R_{1,2})}}$	2.52	$\frac{13}{8} : \frac{13+16}{8+8} \times \frac{2}{2} \mapsto (\frac{13+}{8+})\frac{16+17}{8+1}$
$PE : I_{(R_{1,2})} \times (R_{3,4}) \mapsto I_{\sum_{i=1}^{4} R_i}$	0.34	$\frac{13}{8} : \frac{17+17}{9+1} \times \frac{2}{2} \mapsto \frac{9}{1}$
$PE_{I_{(R_{1,2})}} : (R_{3,4}) \mapsto I_{\sum_{i=1}^{4} R_i}$	0.22	$\frac{13+16+17}{8+8+1} : \frac{2}{2} \mapsto \frac{9}{1}$

way may proceed as:

$$PE \Rightarrow PE_I \Rightarrow PE_{I_{(R_{1,2})}} \Rightarrow I_{(R_{1,2,3,4})}$$

which amounts to 3.81 (= 1.07 + 2.52 + 0.22) seconds. In general, specialising compilers takes more time than specialising interpreters does. However, of course, the specialised compiler compiles a new program fragment faster than the partial evaluator specialises the specialised interpreter with respect to the new program fragment. For instance, 0.50 vs. 0.84 seconds obtaining $I_{(R_{1,2})}$, and 0.22 vs. 0.34 seconds obtaining $I_{\sum_{i=1}^{4} R_i}$.

Due to the limited examples, a more general conclusion about performance cannot be stated in this paper. However, even if we tried more examples, there would still remain certain difficulties in evaluating the results fairly and deriving some definite statements on performance, because the performance of partial evaluation is strongly dependent on the specificity of the subject program. Some evaluation standard and benchmarks need to be established.

6. Another Application

In some applications, a meta-interpreter, I_1, is used to meta-interpret another meta-interpreter, I_2. This can be extended to any level of hierarchy of meta-interpretation, comprising the *tower of interpretation* (Fig.8).

Each time a next level meta-interpreter is given, incrementally from the bottom, it can be collapsed to the bottom level meta-interpreter, by partial evaluation (Fig.9).

At any level N, the collapsed meta-interpreter I^N can be specialised with respect to the object level program that can be run on the N-th meta-interpreter I_N, which in turn can be run on the $(N-1)$-th meta-interpreter, and so on, obtaining I_P^N. Thus, the resultant I_P^N can run on the system far faster than the original object-level program on the tower of interpretation.

This method may offer another kind of incremental compilation, which advances vertically across the object-meta boundary, rather than horizontally on the same level as shown in the last section.

7. Conclusion

This paper presented a self-applicable partial evaluator in Prolog. The partial evaluator is so simple that its self-application can be performed straightforwardly, thereby realising compiler generation and compiler generator generation. We think that, although the partial evaluator is small and minimal in functionality, the success of its self-application is not a trivial result, since it implies another realisation of the important theoretical result relating partial evaluation to compilation in logic language.

An application of the partial evaluator to the incremental compilation of a meta-interpreter specialised with respect to program fragments given incrementally was described. The method will be promising when used in the development of expert systems whose knowledge base is extended incrementally.

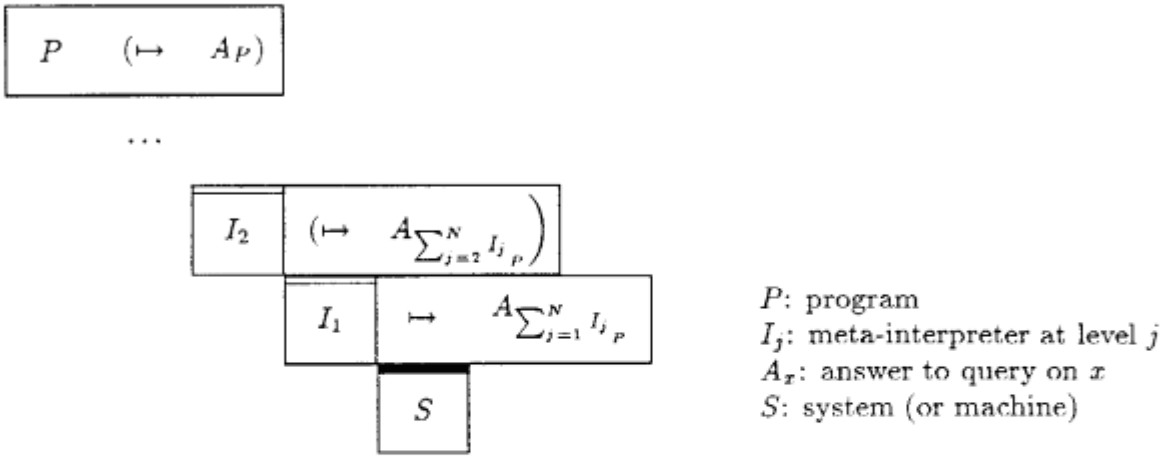


Figure 8 Tower of interpretation

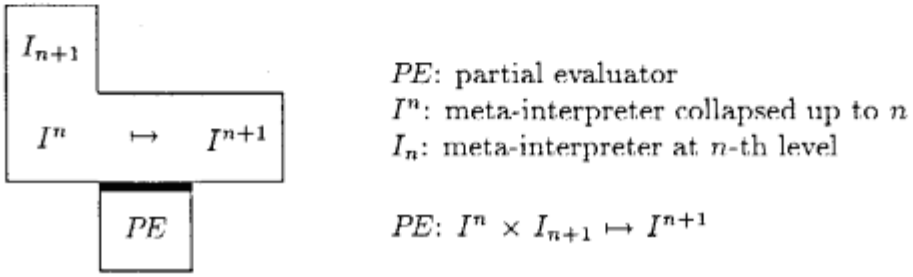


Figure 9 Collapsing the tower of interpretation

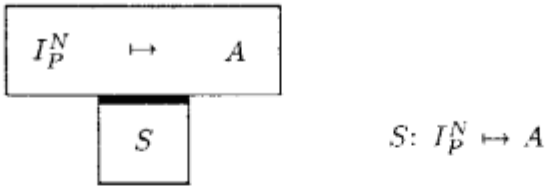


Figure 10 Collapsed meta-interpreter running without overhead

It is desirable to discuss issues concerning automatic derivation of control information such as `$suspend` or `$expand`, and enlargement of the class of the target language from pure Prolog to practical Prolog with negation, cut, `bagof`, etc., thereby making partial evaluation of primitives less tricky. [Naish 85] seems to offer materials closely related to these issues. However, the elaboration of them is beyond the scope of the present paper.

There remain several research themes to be extensively pursued. The first is obtaining a more powerful partial evaluator (that is, applicable to a wide class of source

programs, self-applicable and hopefully fully automatic), after solving those issues stated above. The second is accumulating experience of partial evaluation with practical applications to tune up the algorithm or heuristics. Some of these are being tackled [Fujita 87b] and [Takeuchi 87]. The third is trying the same goals in other languages, especially in parallel logic programming languages such as GHC [Furukawa 87].

Acknowledgements

We would like to express our gratitude to Dr. K. Fuchi, director of ICOT, for giving us the opportunity of doing this research. We also wish to thank Mr. A. Takeuchi for his advice in initiating this present work, members of the First Laboratory of ICOT for their useful discussions, and the referees for their suggestions and comments that helped improve the draft of this paper.

References

[Ershov 78] A.P. Ershov, On the Essence of Compilation, in E.J. Neuhold (ed.): *Formal Description of Programming Concepts*, 391-420, North-Holland, 1978

[Fujita 87a] H. Fujita, On Automating Partial Evaluation of Prolog Programs, ICOT TM-250, 1987 *(in Japanese)*

[Fujita 87b] H. Fujita, An Algorithm for Partial Evaluation with Constraints, ICOT TM-367, 1987

[Furukawa 87] K. Furukawa and A. Okumura, Unfolding Rules for GHC Programs, in A.P. Ershov, D. Bjørner and N.D. Jones (eds.): *Proc. of Workshop on Partial Evaluation and Mixed Computation, Gl. Avernæs, Denmark, October 1987*, North-Holland, 1988 (to appear)

[Futamura 71] Y. Futamura, Partial Evaluation of Computation Process - An Approach to a Compiler-compiler, *Systems, Computers, Controls*, Vol. 2, No. 5, 45-50, 1971

[Futamura 82] Y. Futamura, Partial Evaluation of Programs, in E. Goto, et al. (eds.): *RIMS Symposia on Software Science and Engineering, Kyoto, Japan, 1982*, Lecture Notes in Computer Science, Vol. 147, 1-35, Springer-Verlag, 1983

[Jones 85] N.D. Jones, P. Sestoft and H. Søndergaard, An Experiment in Partial Evaluation: The Generation of a Compiler Generator, in J.P. Jouannaud (ed.): *Rewriting Techniques and Applications*, Lecture Notes in Computer Science, Vol. 202, 124-140, Springer-Verlag, 1985

[Kahn 82] K.M. Kahn, A Partial Evaluator of Lisp Programs Written in Prolog, in M. Van Caneghem (ed.): *First International Logic Programming Conference, Marseille, France*, 19-25, 1982

[Kahn 84] K.M. Kahn and M. Carlson, The Compilation of Prolog Programs without the Use of a Prolog Compiler, in *International Conference on Fifth Generation Computer Systems, Tokyo, Japan,* 348-355, 1984

[Komorowski 82] H.J. Komorowski, Partial Evaluation as a Means for Inferencing Data Structures in an Applicative Language: A Theory and Implementation in the Case of Prolog, in *Ninth ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico,* 255-267, 1982

[Kursawe 86] P. Kursawe, How to Invent a Prolog Machine, in E. Shapiro (ed.): *Third International Conference on Logic Programming, London, United Kingdom, 1986,* Lecture Notes in Computer Science, Vol. 225, 134-148, Springer-Verlag, 1986

[Levi 86] G. Levi, Object Level Reflection of Inference Rules by Partial Evaluation (Extended Abstract), in P. Maes and D. Nardi, (eds.): *Workshop on Meta-Level Architectures and Reflection, Sardinia, October 1986* (to appear)

[Naish 85] L. Naish, Negation and Control in Prolog, Lecture Notes in Computer Science, Vol. 238, Springer-Verlag, 1985

[Pereira 79] L.M. Pereira, F.C.N. Pereira and D.H.D. Warren, User's Guide to DECsystem-10 PROLOG, Occasional Paper 15, Dept. of Artificial Intelligence, Edinburgh, 1979

[Safra 86] S. Safra and E. Shapiro, Meta Interpreters for Real, in H.J. Kugler (ed.): *Information Processing 86, Dublin, Ireland,* 271-278, North-Holland, 1986

[Sestoft 86] P. Sestoft, The Structure of a Self-Applicable Partial Evaluator, in H. Ganzinger and N.D. Jones (eds.): *Programs as Data Objects, Copenhagen, Denmark, 1985,* Lecture Notes in Computer Science, Vol. 217, 236-256, Springer-Verlag, 1986

[Takeuchi 86] A. Takeuchi and K. Furukawa, Partial Evaluation of Prolog Programs and Its Application to Meta Programming, in H.J. Kugler (ed.): *Information Processing 86, Dublin, Ireland,* 415-420, North-Holland, 1986

[Takeuchi 87] A. Takeuchi and H. Fujita, Competitive Partial Evaluation – Some Remaining Problems of Partial Evaluation, in A.P. Ershov, D. Bjørner and N.D. Jones (eds.): *Proc. of Workshop on Partial Evaluation and Mixed Computation, Gl. Avernæs, Denmark, October 1987,* North-Holland, 1988 (to appear)

[Vasey 86] P. Vasey, Qualified Answers and Their Application to Transformation, in E. Shapiro (ed.): *Third International Conf. on Logic Programming, London, United Kingdom,* Lecture Notes in Computer Science, Vol. 225, 425-432, Springer-Verlag, 1986

[Venken 84] R. Venken, A Prolog Meta-interpreter for Partial Evaluation and Its Application to Source to Source Transformation and Query-optimization, in T. O'Shea (ed.): *ECAI-84, Advances in Artificial Intelligence, Pisa, Italy,* 91-100, North-Holland, 1984